Fall 2015

Implementing Type Inference in Jedi

Vaibhav Kamble
San Jose State University

Implementing Type Inference in Jedi

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Vaibhav Kamble

December 2015

The Designated Project Committee Approves the Project Titled

Implementing Type Inference in Jedi

by

Vaibhav Kamble

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Jon Pearce	Department of Computer Science
Cay Horstmann	Department of Computer Science
Thomas Austin	Department of Computer Science

ABSTRACT

Implementing Type Inference in Jedi

by Vaibhav Kamble

This thesis begins with an overview of type systems: evolution, concepts, and problems. This survey is based on type systems of modern languages like Scala and Haskell. Scala has a very sophisticated type system that includes generics, polymorphism, and closures. It has a built-in type inference mechanism that enables the programmer to exclude certain type annotations. It is often not required in Scala to mention the type of a variable because the compiler can infer the type from the initialization of the variable. Study of such type system is demonstrated by the implementation of the type system.

A type system for a demonstration language 'Jedi' is developed. It includes a runtime type identification system for Jedi values and a static type system for Jedi expressions. Jedi features include arithmetic, logic, and declarations. It also supports lambda expressions, blocks, and closures. In Jedi, each value and expression have a type, and types are also values.

Challenges faced in the Jedi type system include determining the types of complex expressions such as lamdas, function calls, and nested expressions. A Jedi reference interpreter and its type system are implemented in Scala.

ACKNOWLEDGMENTS

I would like to thank my project advisor Dr. Jon Pearce for his constant guidance and for also believing in me. Without his mentoring and support, this project would not have been achieved. Also, I would want to thank him for his patience throughout the process.

I would also like to thank the committee members Prof. Cay Horstmann and Prof. Thomas Austin for their valuable inputs and to observe the progression of the project closely.

Moreover, I would like to thank my family and friends for being there with me and encouraging me throughout my masters program.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

Overview of Type Theory and Type Checking

Type systems are responsible for detecting and assigning types to values and expressions in programming languages. Static type system determines the types of expressions occurring in a program before the program is executed. Thus, type errors like use of a string in a context where a number is expected can be detected and fixed before the program is operational.

Type systems are expected to provide the following essential functions.

- **Safety:** Type checking should prevent runtime failures by detecting invalid executions of functions and operations.
- **Optimization:** Compilers and interpreters can get useful information from type checkers. This information can be used for code optimization. For example type checking provides details of the required storage for a particular data type.
- **Documentation:** Documentation on type constructs helps developers to understand how to use them.
- **Abstraction:** The programmer can think a higher level of abstraction in programming with the capability of naming types and hiding their implementation details. Such data hiding enables modeling of the problem domain and allows implementation change of types and their operations without having any impact on the correctness of programs.[1]

1.1 History of type checking

Type is associated with every value generated by a program, either explicitly or implicitly. In a strongly typed language, the language processor provides a type checker that ensures that type errors will not occur at run time. So, strongly typed languages are statically type checked. Dynamic type checking normally occurs during program execution, while static type checking occurs before program execution, typically at compile time. Other type related checks may take place at program link time.[2]

Dynamically typed languages like LISP and Scheme, perform type checking at runtime. Thus, code for an addition operation may have to check the type of its parameters just before the operation is performed. In statically typed languages, the type of each expression in a program that is the type of value the expression will produce when executed, is determined during compilation. Dynamically typed programming languages can be more flexible and expressive than statically typed languages because the type checking is postponed until run time. Consequently, all static type checkers are incomplete. Programs that might run without problems may be rejected by a static type checker.[2]

Programmers familiar with dynamically type checked languages may worry that the utilization of a static type system will restrict the use of programs that can be dynamically determined type safe. For example, the static type system of Pascal is so inflexible that it will not allow programmers to write a single sort procedure that will work for integer arrays of different sizes, let alone for arrays of other types such as reals or characters. C also has a similarly restrictive type system, but it provides mechanisms of type casting that allows the programmers to avoid the static type system when it gets in the way.[2]

However, Modern programming languages such as Java and Haskell, support various forms of polymorphism (discussed below), which restores much of the flexibility found in dynamically typed languages.

1.2 Ad hoc polymorphism

Ad hoc polymorphism, also called overloading, allows different functions to share a name provided their parameter lists are different. For example, there can be many functions named add: integer addition, floating point addition, string concatenation, etc. Overloading makes a library easier to use because users don't need to remember different names for closely related functions.[4]

So, In function overloading the function is uniquely recognized not by its name, but by the organization of its name and the number, order, and types of its arguments. Object-oriented languages support this type of polymorphism by allowing an operator to be overloaded similar to functions. Languages that are not dynamically typed and need ad hoc polymorphism have longer function names such as appendInt, appendString. This provides an advantage as function names are more descriptive and provides disadvantage as function names are overly verbose.[4]

1.3 Subtype polymorphism

If S is a subtype of T denoted S <: T then any expression of type S can be reliably used in a context where an expression of type T is expected. A type may be a subtype of many types. Subtyping is a type polymorphism. Subtyping is associated to the idea of bounded quantification in mathematical logic.[4]

There are two common subtyping scheme - nominal subtyping and structural subtyping. In nominal subtyping, types are listed in a particular way may be subtypes

of each other, and in structural subtyping, construction of two types determines whether or not one is a subtype of the other. In structural subtyping, if objects of type A can manage all of the messages that objects of type B can manage, then A is a subtype of B despite whether either inherits from the other. Languages with subtyping features fall into two general classes: inclusive implementation and coercive implementation. In the inclusive implementation, any value of type A also denotes the same value of type B if A<: B. The subtyping implemented by subclassing in an object-oriented language is inclusive. In coercive implementations, any value of type A can be automatically changed into one of type B. Subtyping relations that associate integers and floating-point numbers, which are expressed differently, are coercive.[4]

Types of record i.e. compound data like struct or tuple use width and depth subtyping. These are the two ways of getting a new type of record that allows the same operations as the original record type. Record subtype supports the similar operations on the fields as the original type supported.This can be achieved by width subtyping, which adds more fields to the record. This ensures, every field appearing in the width supertype will appear in the width subtype. So, operations allowed on supertype will be supported for the subtype. In the case of depth subtyping, various fields are replaced with their subtypes. So, every operation carried for a field in the supertype is supported for its subtype, any operation possible on the record supertype is supported by the record subtype.[4]

In the case of function types, function type S1 $\rightarrow$ S2 is a subtype of function type T1 $\rightarrow$ T2 if T1 $<:$ S1 and S2 $<:$ T2. The argument type of S1 $\rightarrow$ S2 is said to be contravariant because the subtyping relation is inverted for it, whereas the return type is covariant. This happens because the refined type is more generous in the types it accepts and more cautious in the type it returns. In languages allowing side

effects, subtyping is not enough to ensure that a function can be reliably used in the context of another.[4]

1.4 Parametric polymorphism

In parametric polymorphism, a function or a data type are generically defined to handle values in a similar way without depending on their type. Such functions and data types are forms of the basis of generic programming and known as generic functions and generic datatypes respectively.[4]

For example, consider a function append that joins two lists can be constructed without considering the type of elements in the list. Such function can append lists of real numbers, lists of strings, and so lists of other data types. So, lets assume T is a type variable and denote the type of elements in the lists. Then append can be typed for all T. [T] $\times$ [T] $\rightarrow$ [T], where [T] denotes the type of lists containing elements of type T. We can conclude that the type of append is parameterized by T for all values of T. For each place where append is applied, a value is decided for T.[4]

Parametric polymorphism contrasted with ad hoc polymorphism because in ad hoc polymorphism a single polymorphic function can have some separate and heterogeneous implementations depending on its arguments types. Thus, ad hoc polymorphism supports a limited number of such distinct types, as each type has to provide its special implementation.[4]

1.5 Hindley–Milner type system

Hindley–Milner is a classical type system for the lambda calculus with parametric polymorphism, first presented by J. Roger Hindley and then rediscovered by Robin Milner. Luis Damas added a close formal analysis and proof of the method in his

Ph.D. thesis. Hindley-Milner is efficient and widely accepted due to its completeness and its ability to infer the most general type of a delivered program without the need of any type annotations or other hints provided by the programmer. Algorithm W is a fast algorithm, producing type inference in almost linear time with the size of the source, making it effectively usable to type large programs. HM is preferably used for functional languages. It was first completed as part of the type system of the programming language ML. Since then, HM has been extended in various ways, most notably by constrained types as used in Haskell. There is even some research being tried to find a way to apply the power of Hindley-Milner to optimize dynamic languages like Ruby, JavaScript, and Clojure.[5][6]

Hindley-Milner deductive system describes how expressions and types fit each other. A deductive system consists of the assumptions and rules of inference that can be used to create the theorems of the system. It supports different ways to come to a conclusion and same expression can have different types, dissimilar conclusions about an expression are plausible. Opposite to this, the type inference method itself (Algorithm W) is described as a deterministic step-by-step method, leaving no choice what to do next. Thus clearly, conclusions not present in the logic might have been made building the algorithm, which require a closer look and reasons but would perhaps remain non-obvious without the above differentiation.[6]

In this thesis work, we have referred HM rules for deducing statements about type inference to define rules for type system of demo language Jedi. More details on Jedi are in the next chapters.

1.6 Type checking in Haskell

In Haskell, an expression evaluates to a value and has a static type. Values and types are not mixed in Haskell. The type system of Haskell allows user-defined datatypes and permits not only parametric polymorphism using a conventional Hindley-Milner type structure but also ad hoc polymorphism or overloading. Haskell types are strong, static, and can be automatically inferred.[7]

Haskell has a strong type system. It ensures that a program is free from certain kinds of errors. For example, if a function requires an argument of data type integers, and if it receives an argument of type string, then the compiler will reject it. Also, Haskell does not automatically coerce i.e.cast values from one type to another. C compilers will implicitly cast a value of type int into a float if a function requires a parameter of type float. But Haskell compilers will result in a compilation error. Casting should be done explicitly by applying coercion functions, which would result in low performance. The advantage of strong typing is that it detects real bugs in code before they can result in failure.[7]

Haskell has a static type system that enables the compiler to know the type of each value and expression at the time of compilation before any code is executed.

ghci> True && “false"

<interactive>:1:8: Couldn’t match expected type ‘Bool’ against inferred type ‘[Char]’ In the second argument of ‘(&&)’, namely “‘false"’ In the expression: True && "false" In the definition of ‘it’: it = True && "false"

We can deduct from above error message in Haskell that he compiler has deduced that the type of the expression "false" is Char. The && operator expects its operands to be of type Bool, but the type of "false" does not equal the expected type, the

compiler denies this expression as ill-typed. Haskell's strong and static typing makes it difficult for type errors to occur at runtime. This ensures that once code compiles, it's more likely to work perfectly than in other languages.[7]

Finally, a Haskell compiler can implicitly infer the types of every expression in a program. This is known as type inference.

1.7 Type checking in Scala

Scala is a statically typed language. Its type system is one of the most sophisticated in a modern programming language. Scala type system supports variety annotations, upper and lower type bounds, inner classes, generic classes, and abstract types, compound types, explicitly typed self-references, implicit parameters, and views, polymorphic methods.[8]

Scala has built-in support for classes parameterized with types. Generic classes are useful to develop collection classes. It supports variance annotations of type parameters of generic classes. Variance annotations are added when a class abstraction is defined. Type parameters and abstract types have constraints of type bound. Type bounds limit the values of the type variables and expose more information about the members of such types. An upper type bound T <: A states that type variable T refers to a subtype of type A. While upper type bounds narrow a type to a subtype of another type, lower type bounds represent a type to be a supertype of another type. The term T >: A proves that the type parameter T or the abstract type T refer to a supertype of type A.[8]

Scala supports Abstract types. Abstract types are types whose status is not precisely known. In many cases object of the class has a type member T, but the definition of the class does not reveal to what concrete type the member type T

corresponds. So such type definitions can be overridden in subclasses. This allows us to expose more information about an abstract type by stretching the type bound. Sometimes it is required to show that the type of an object is a subtype of some other types. In Scala, this can be proved with the help of compound types, which are intersections of object types.Compound types can consist of many object types, and they may have a single refinement that can be used to fine the signature of existing object members. Polymorphic methods are supported in Scala. Functions in Scala can be parameterized with both values and types. The type system of Scala can infer types of method argument. This is done by scanning at the types of the provided value parameters and at the context where the method is called.[8]

Scala has a built-in type inference mechanism that allows the programmer to neglect certain type annotations. It is not necessary for Scala to specify a variable since the compiler can infer the type from the expression of the variable. Also, return types of methods can be excluded since they correspond to the type of the body, which gets deduced by the compiler. For recursive methods, the compiler is unable to understand a result type. As mentioned above, in the case of polymorphic methods or generic classes, it is not mandatory to specify type parameters. The Scala compiler infers such absent type parameters from the context and the types of the method parameter values.[8]

1.8 Type checking in C++

C++ is statically typed language. Its compiler statically determines whether operations performed on variables of the correct type. The C++ type system is extensible in nontrivial ways aiming for equal support for built-in types and user-defined types.[9]

C++ type-checking and data-hiding features depend on compile-time analysis of programs to prevent accidental corruption of data. C++'s static type system is flexible, and the use of simple user-defined types implies little, if any overhead. The aim is to support a style of programming that represents distinct ideas as distinct types, rather than just using generalizations, such as integer, floating-point number, string, "raw memory," and "object," everywhere.[9]

C++ compilers and development tools support such type-based analysis. Maintaining most of C as a subset and preserving the direct mapping to the hardware needed for the most demanding low-level systems programming tasks implies the ability to break the static type system.

CHAPTER 2

Overview of Sith

2.1 Sith

Sith is a programming language developed by Jon Pearce to teach students how to write parsers and reference interpreters. Following are some features of Sith language, demonstrated with help of an interpreter.[10]

- **Declaration** - The 'def' syntax allows the user to declare new variables and store it in the environment to retrieve and use whenever required. It creates an identifier - value mapping in the environment. Lookup in environment map is required to get a value of defined variable.We will discuss more about environment in further part of chapter.[10]

```
-> def x = 1
OK
-> x
1.0
```

- **Arithmetic operations** - Sith supports arithmetic operations like addition, subtraction, multiplication and division. Like any other programming language, Sith requires arithmetic operations for computations. Sith can conduct series of arithmetic operation considering operator precedence. Arithmetic operations are supported on data types like Number and Rational. For rest of the invalid data types, arithmetic operation throws an error.[10]

```
-> 1 + 2
3.0
-> 2 * 3
6.0
-> 6 / 3
2.0
```

- **Logical operations** - Conjunction and Disjunction operations are supported in Sith. Logical operation results in boolean values. The logical operation expressions return a 'false' or 'true' result over a conditional expression. There are three types of logical operators, each of which is concerned with conditional expressions. These are : && , || , < , ==.

  All of these logical operators have a distinct effect on the conditional expressions.[10]

```
-> true && false
false
-> true || false
true
-> (1<2) && (3==3)
true
```

- **The conditional operation** - 'if' statements are supported in Sith, which checks condition in if clause and execute appropriate expression. Conditional operations perform different computations depending on whether a specified boolean condition evaluates to true or false. Following is the interpreter console that demonstrate how to use 'if-else' statement in Sith.[10]

```
-> if(true) 1 else 2
1.0
-> if(1<2) 1 + 2 else 1 / 2
3.0
```

- **Tuple** - It is a limited ordered list of elements. Sith has compound data types like tuple, and it also provide instructions to access tuple elements. Tuple elements can be any primitive data types of Sith or compound data type as a tuple. Following interpreter console show, how tuple is defined in Sith and 'get' is the method to access tuple elements by their index.[10]

```
-> def a = Tuple(1,true)
OK
-> a
(1.0 * true)
-> get(a,1)
true
```

- **Lambda expressions** - These are one the dominant feature of Sith. A lambda expression is an anonymous function that you can use to create delegates or expression tree types. Sith lambda takes input parameters with type information. And it has expression i.e. block of statements as the body. One can call/execute lambda by passing parameters of expected data type. Lambda expressions are a function. So, they can be executed like function calls. Following interpreter show, how to define and use a lambda in Sith.[10]

```
-> def a = lambda(x:Number,y:Number) x + y * 3
OK
-> a(2,3)
11.0
```

- **Variables** - These are values, just like numbers and booleans. Sith variables hold values. Following console output show, how to define variables. There is the special syntax to dereference such variable.[10]

```
-> def b = var(1 + 2)
OK
-> b
Variable(3.0)
-> [b]
3.0
```

- **Block** - Sith block allows several expressions to be grouped into a single expression. Blocks also enable programmers to define identifiers with restricted scopes and lifetimes. Like many programming language Jedi blocks, groups statements in curly braces i.e.statement 1; statement 2; statement 3. Following console output demonstrate, how blocks are defined and executed in Sith.[10]

```
-> def small = {def delta = 10; delta<1}
OK
-> small
false
```

Unfortunately, Sith lacks dynamic and static type checking. That makes Sith prone to run-time type errors. Expressions are getting executed without considering the type of argument expression which may lead Sith programs to failure. But with a sophisticated type system this can be avoided. Jedi, an advanced version of Sith has all capabilities of Sith along with a type checking mechanism that makes it type safe. More information about Jedi[10] is discussed in the next chapter.

2.2 Sith language design

Value is one of the basic building blocks of Sith. Every expression produces a value when executed. Sith values include numbers, booleans, closures, variables, and environments.

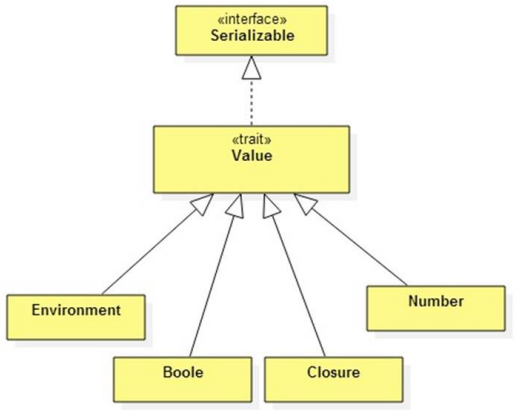

Figure 1: Jedi Values

Expressions are program elements that can be executed. Executing an expression produces a value. Executing an expression requires an environment which specifies bindings between identifiers (names) and values.[10] Literal, Identifier, FunCall and SpecialForm are expressions. On execution literal simply returns the value associated with it. Identifier- value mapping is stored in environment. Environment lookup carried out to return value on identifier execution.

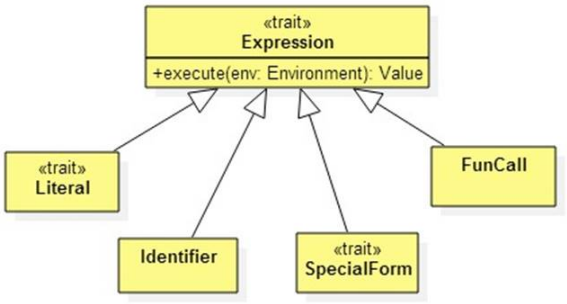

Figure 2: Jedi Expressions

On execution literal simply returns the value associated with it. Identifier - value mapping is stored in environment. Environment lookup carried out to return value on identifier execution.

As earlier mentioned, 'SpecialForm' is an expression. But what are the special

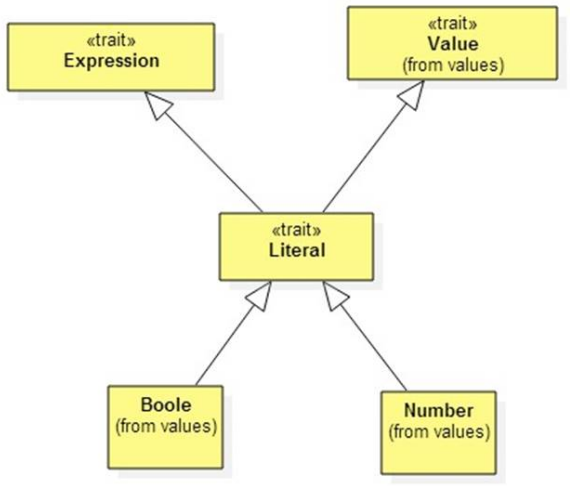

Figure 3: Jedi Literal

forms of expressions? Sith supports logical and conditional operations. So, Conditional, Conjunction, Disjunction, and Declaration are the special form of expression to represents conditional and logical operations. Below figure shows, Declaration is a special form of expression, and it requires Identifier which is another expression. Conjunction and Disjunction need multiple expressions as operands.[10]

Conjunction and disjunction use a special execution algorithm called short-circuit evaluation: evaluate operands from left to right until the answer is known. Conditional evaluation evaluates the condition, then either evaluates the consequence or the alternative but not both.

Sith lambda expressions are local functions that can be passed as arguments or returned as the value of function calls. Lambda is considered as 'SpecialForm' in Sith, which takes identifiers as a parameter and has the expression as a body. Sith block allows several expressions to be grouped into a single expression.

Sith Iteration and Assignment are also special forms of expression. 'UndefinedException', 'SyntaxException' and 'TypeException' are exception classes in Sith.

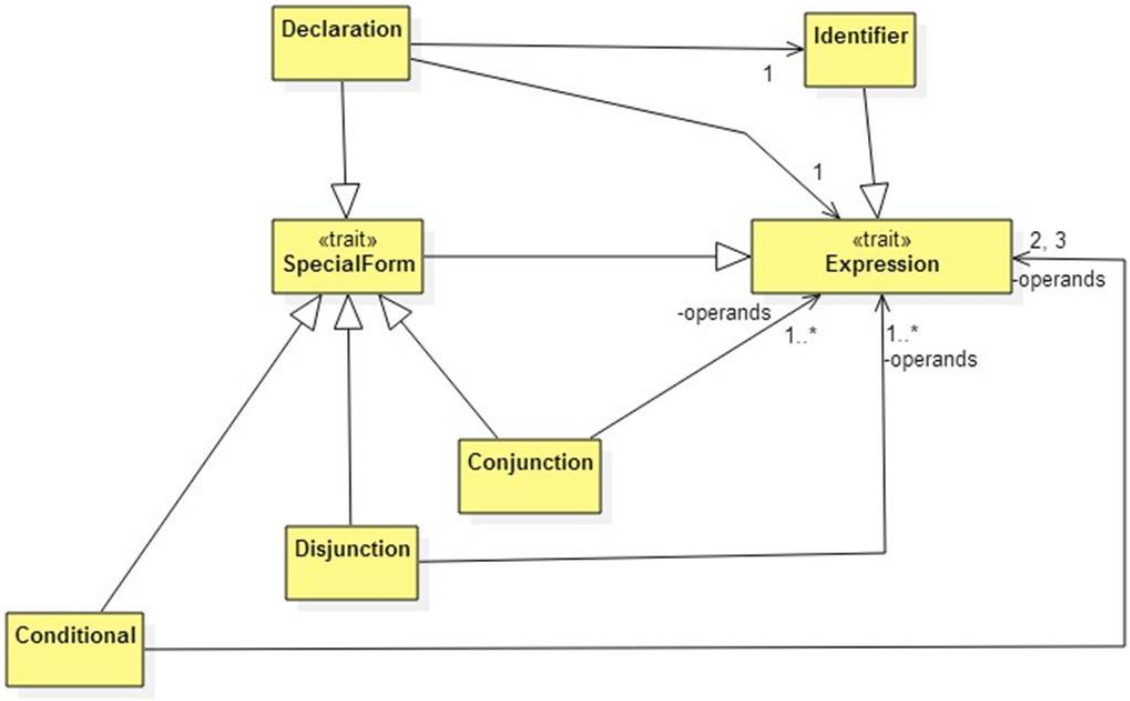

Figure 4: Expression - Special Form

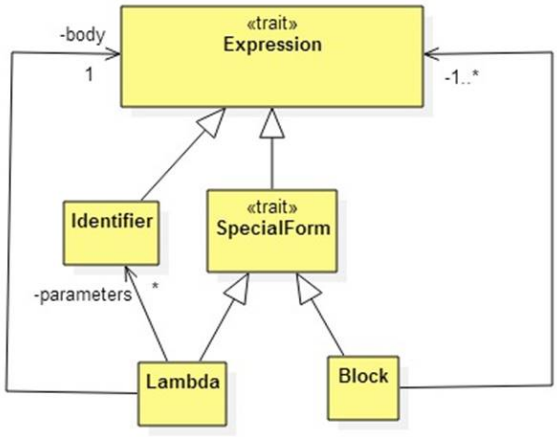

Figure 5: Lambda - Special Form

2.3 Scala - Meta Language

Sith interpreter is implemented in Scala. Following are some useful features of Scala.

- **Object-oriented** - Scala is an object-oriented language. Each value in Scala is an object, and every operation is considered as method-call. The language supports classes and traits as a part of the advanced component architecture. Scala supports many traditional design patterns. For example, object definitions support singleton design pattern, and visitor pattern is similar to pattern matching in Scala. Scala enables users to add new operations to existing classes.[8]

- **Functional** - Even though its syntax is somewhat conventional, Scala is also a functional language. It has everything you would require, including first-class functions, a library with adequate immutable data structures, and a general inclination of immutability over mutation. Unlike with many traditional functional languages, Scala allows a gradual, easy migration to a more functional style. Over time, one can progress to gradually eliminate mutable state in your applications, phasing in safe, functional composition patterns instead.[8]

  Functional PL, Scala allows programmers to define combinators and that it has a library that treats the non-terminals in an EBNBF as parser functions and EBNF operations such as $\sim$, |, and * as combinators that create complex parsers from simpler ones.

- **Seamless Java Interoperability** - Scala runs on the JVM. Java and Scala classes can be freely combined, no matter whether they remain in different projects or the same. They can even mutually relate to each other. Java libraries, frameworks, and tools are all accessible. Tools like Maven, Eclipse, IntelliJ, or Netbeans, frameworks like Spring or Hibernate all work with Scala. Scala runs on all common JVMs.[8]

CHAPTER 3

Overview of Jedi

Jedi is an extension of Sith language. It has all the capabilities of Sith, plus it has type checking, which makes Jedi efficient and secure. To enable type checking, Types introduced in Jedi. In Jedi, every value and expression have a type. In fact, Jedi types are Values, which we will discuss more in next chapters. Jedi supports primitive data types like Boolean and Number. It also supports compound data types like TupleType and FunType.

Jedi has an efficient type system. It ensures that a program is free from certain kinds of errors. These errors may occur due to invalid expressions. Jedi has a static type system that enables the language processor to know the type of every value and expression before any code execution, and this enables Jedi to detect potential type violations. Such error detection, prevent runtime failure of the program.

3.1 Type function

Type is the utility command used to determine the type of values and expressions in Jedi. Using type function, one can inquire for the type of Jedi expression and value. It serves as a primitive type of reflection in Jedi. Reflection is a language's ability to inspect expressions and values.

Following is the syntax for type command in Jedi

-> type <expression/value>

3.2 Type system demo

In this chapter, we will demonstrate the features of Jedi type system.

3.2.1 Type checking of Jedi values

```
-> type 1
Number
-> type 1.2
Number
-> type true
Boolean
-> type false
Boolean
```

Jedi type system infers the type of 1 is Number and true is Boolean.

3.2.2 Type of arithmetic and logical operation

```
-> type 1 + 2
Number
-> type 1 + (2/3)
Number
-> type 1<2
Boolean
-> type 2<3 && 4<3
Boolean
```

The Jedi type checker determines the types of values and verifies them against the type rule of arithmetic and logical operation.

3.2.3 Type of identifier

```
-> def a = 1
OK
-> a
1.0
-> type a
Number
-> type a + 1
Number
-> def b = a
OK
-> type b
Number
```

Type checker determines the type of identifier by identifying the type of value or expression it is bound. This is called as type inference.

3.2.4 Type of Tuple

```
-> def a = Tuple(1, 2, true)
OK
-> a
(1.0 * 2.0 * true)
-> type a
(Number * Number * Boolean)
-> def b = Tuple(1, a)
OK
-> b
(1.0 * (1.0 * 2.0 * true))
-> type b
(Number * (Number * Number * Boolean))
-> get(b, 1)
(1.0 * 2.0 * true)
-> type get(b, 1)
(Number * Number * Boolean)
```

A tuple is a limited ordered list of elements. Type of tuple is composed of a type of each element of the tuple. Tuple(a,b,c) has a type (type of a × type of b × type of c).

3.2.5 Type of Variable

```
-> def a = var(1)
OK
-> a
Variable(1.0)
-> type a
Variable(Number)
-> type [a]
Number
```

Type of variable is composed of its content type. So, type of variable is 'variable(type of variable content)'.

3.2.6 Type of Block

```
-> def delta = 1
OK
-> def small = {def a = 3;lambda(x:Number) x + delta < a}
OK
-> small(1)
true
-> small(2)
false
-> type small
((Number) -> Boolean)
-> type small(1)
Boolean
```

A Jedi block allows several expressions to be grouped into a single expression. Type of block can be determined by the type of the last expression in the block.

3.2.7 Type of iteration

```
-> type while(true) 1 + 2
Number
-> type while(true) {if(true) 1 else 4}
Number
-> while(1) 6 + 6
Error
```

Type of Jedi iteration can be an error or any valid Jedi type. If an iterative instruction has a logical condition of type error, then the type of expression is an error. Otherwise, type of expression equals to the type of expression body.

3.2.8 Type of conditional expression

```
-> type if(1) true else false
Error
-> type if(true) 1 else false
Error
-> def a = 1
OK
-> a
1.0
-> type if(a<5) 1+2 else a*4
Number
```

The type of a conditional expression is an error if the type of condition expression is not boolean and consequent and alternate expressions are of different. Otherwise, type of conditional expression is equal to consequent or alternate expression type.

3.2.9 Type of lambda expression

```
-> def a = lambda(x:Number) x < 1
OK
-> type a
((Number) -> Boolean)
-> type a(1)
Boolean
-> def y = lambda(a:Boolean, b:Boolean) if(a && b) 1 else 2
OK
-> type y
((Boolean * Boolean) -> Number)
```

Changes made in the lambda expression to enable type checking for lambda. In Jedi, Lambda expressions take parameters with their types as input. Type of lambda is inferred by its domain type and range type.

3.2.10 Type of Jedi type

```
-> type Number
Type
-> type Boolean
Type
```

Everything in Jedi has the type, even types have their type. By this rule, number and boolean are of type 'Type', which indicates those are the data types.

CHAPTER 4

Dynamic Type Checking in Jedi

Jedi type system has primitive types and compound types. Following class diagram shows the structure of Jedi type system.

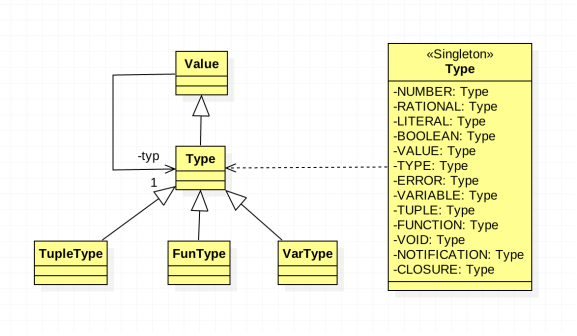

Figure 6: Jedi - Type Hierarchy

Every type in Jedi is a value, and each value has a type. TupleType, FunType and VarType are the compound types in Jedi. Companion object Type represents the primitive types of Jedi.

4.1 Primitive types

Primitive types are the most fundamental data types of Jedi; these include NUMBER, RATIONAL, LITERAL, BOOLEAN, VALUE, TYPE, ERROR, VARIABLE, TUPLE, FUNCTION, VOID, NOTIFICATION and CLOSURE. These types serve as the building blocks of data manipulation in Jedi.

Following are the formation rules for primitive types in Jedi.

- NUMBER type represents numerical values.

$$\overline{\text{NUMBER is a Type}}$$

- BOOLEAN type represents boolean values i.e. true or false.

$$\overline{\text{BOOLEAN is a Type}}$$

- RATIONAL type denotes numerical values and is a subtype of NUMBER.

$$\overline{\text{RATIONAL is a Type}}$$

- LITERAL type represents literal values in Jedi.

$$\overline{\text{LITERAL is a Type}}$$

- VALUE is a default type of all values.

$$\overline{\text{VALUE is a Type}}$$

- ERROR type denotes an error in Jedi. Expressions and values with ERROR types result in failures on execution.

$$\overline{\text{ERROR is a Type}}$$

- VARIABLE is default type of compound type VarType.

$$\overline{\text{VARIABLE is a Type}}$$

- TUPLE is default type of compound type TupleType.

$$\overline{\text{TUPLE is a Type}}$$

- FUNCTION is default type of compound type FunType.

$$\frac{}{\text{FUNCTION is a Type}}$$

- VOID type

$$\frac{}{\text{VOID is a Type}}$$

- NOTIFICATION type denotes types of declarative instructions.

$$\frac{}{\text{Notification is a Type}}$$

- CLOSURE represents closure values.

$$\frac{}{\text{CLOSURE is a Type}}$$

- TYPE represents Jedi type. Jedi types are value, and they also have their type, which is TYPE. Hence, TYPE is Jedi type.

$$\frac{}{\text{TYPE is a Type}}$$

Following is the code snippet of companion object of class Type, which represents primitive types of Jedi.

```
object Type{
    val NUMBER = new Type("Number")
    val RATIONAL = new Type("Rational")
    val LITERAL = new Type("Literal")
    val BOOLEAN = new Type("Boolean")
    val VALUE = new Type("Value")
    val TYPE = new Type("Type")
    val ERROR = new Type("Error")
    val VARIABLE = new Type("Variable")
    val CLOSURE = new Type("Closure")
```

```
11      val TUPLE = new Type("Tuple")
12      val FUNCTION = new Type("Function")
13      val VOID = new Type("Void")
14      val NOTIFICATION = new Type("Notification")
15  }
```

Listing 4.1: Type companion object

4.1.1 Subsumption Rule

Subsumption rules define that any expression or value of type T may also be given type S if T<:S i.e. if T is subtype S.

$$\frac{\text{x is of type T, T is a subtype of S}}{\text{x is of type S}}$$

Subtyping allows term of one type to get used in a context where a term of another type is expected. In, such case that type is a subtype of another type. If T is a subtype of S i.e. T <: S, then any term of type S can be reliably used in a context where a term of type T is expected.

```
-> def a = Rat(1,2)
OK
-> type a
Rational
-> def b = 1
OK
-> type b
Number
-> def fun = lambda(x:Number) x + 1
OK
-> fun(b)
2.0
-> fun(a)
1.5
-> fun(true)
Error
->
```

Figure 7: Example : Primitive subtyping - Rational masquerading as Number

Above example demonstrates, Rational value is masquerading as Number value in lambda function. Hence, Rational is subtype of Number.

Subtyping rule explains, subtyping relation between Jedi primitive types.

- Two equal types are considered as subtypes of each other i.e. S <: T if S = T.

$$\frac{\text{type T equals type S}}{\text{T is a subtype of S, S is a subtype of T}}$$

- ERROR type is subtype all Jedi types.

$$\frac{}{\text{ERROR is a subtype of all Types}}$$

- RATIONAL type is a subtype of NUMBER type.

$$\frac{}{\text{RATIONAL is a subtype of NUMBER}}$$

Following code snippet is code interpretation of Jedi primitive type subtyping rule.

```
1   def subType(other:Type): Boolean = {
2     if(this == other)
3       return true;
4     if(this ==Type.ERROR)
5       return true
6     if(this == Type.RATIONAL && other == Type.NUMBER)
7       return true
8     return false
9   }
```

Listing 4.2: Primitive type subtyping

4.2 Compound types

Compound types in Jedi are constructed using primitive types. TupleType, FunType and FunType are compound types in Jedi.

4.2.1 Tuple Types

A tuple is a finite ordered list of elements or components. TupleType denotes the type of Tuple.

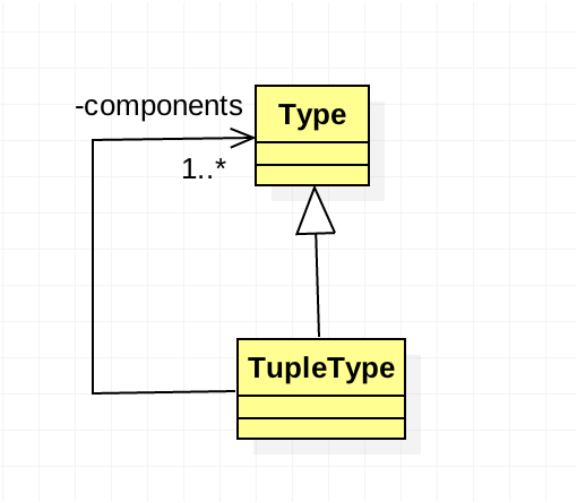

Figure 8: Tuple type - class diagram

Class diagram shows that TupleType is composed of its components types.

Following is the code snippet for TupleType construct. Refer appendix for complete code.

```
1 class TupleType(components: List[Type]) extends Type { ... }
```

Listing 4.3: TupleType class

Formation rule for TupleType is,

If T1 is a type, T2 is a type and Tn is a type then T1 × T2 × . . . Tn is a type

$$\frac{\text{T1 is a type, T2 is a type, ... Tn is a Type}}{\text{T1} \times \text{T2} \times \text{... Tn is a type}}$$

TupleType subtyping rule explains, subtyping relation between two TupleTypes.

If T1 is a subtype of S1 and T2 is a subtype of S2 then (T1 $\times$ T2) is a subtype of (S1 $\times$ S2).

$$\frac{\text{T1 is a subtype of S1, T2 is a subtype of S2}}{\text{(T1} \times \text{T2) is a subtype of (S1} \times \text{S2)}}$$

Code snippet is a interpretation of TupleType subtyping rule.

```
override def subType(other:Type): Boolean ={
 if(other==Type.TYPE)
   return true
 if(!other.isInstanceOf[TupleType])
   return false;
 val tupleType = other.asInstanceOf[TupleType]
 if(tupleType.getComponent().size==components.size){
    for((m,n) <- components.zip(tupleType.getComponent())){
      if(!m.toString().equals(n.toString()) && !m.subType(n)){
        return false
      }
    }
  }
  else{
        return false
  }
 return true
}
```

Listing 4.4: TupleType subtyping

```
-> def a = Tuple(1,2)
OK
-> def b = Tuple(3,Rat(4,5))
OK
-> type a
(Number * Number)
-> type b
(Number * Rational)
-> def fun = lambda(x:Tuple(Number,Number)) x
OK
-> fun(a)
(1.0 * 2.0)
-> fun(b)
(3.0 * 0.8)
```

Figure 9: Example : TupleType subtyping

Example demonstrates, (Number × Rational) tuple is masquerading as (Number × Number) tuple in lambda function.

4.2.2 Function Types

FunType represents the type of function. It is composed of domain and range. Domain represents the function parameter type, and the range is the type of function expression.

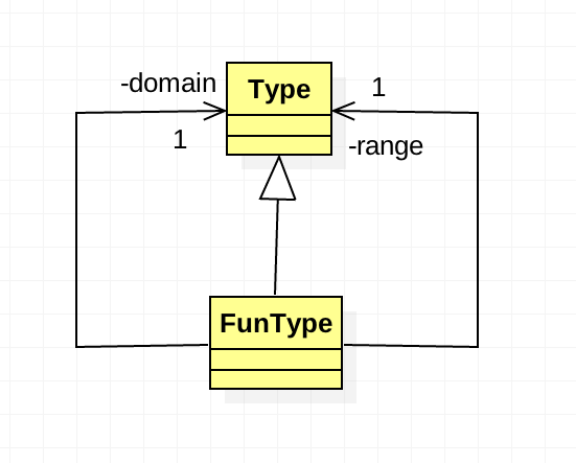

Figure 10: FunType - class diagram

Class diagram shows that FunType is composed of function domain type and range type.

Following is the code snippet for FunType construct. Refer appendix for complete code.

```
class FunType(domain: Type, range:Type) extends Type{ ... }
```

Listing 4.5: FunType class

Formation rule for FunType is,

If T1 is a type, T2 is a type then T1 → T2 is a type.

$$\frac{\text{T1 is a type, T2 is a type}}{\text{T1} \rightarrow \text{T2 is a type}}$$

FunType subtyping rule explains, subtyping relation between two FunType.

If T3 is a subtype of T1 and T2 is a subtype of T4 then (T1 → T2) is a subtype of (T3 → T4).

$$\frac{\text{T3 is subtype of T1, T2 is subtype of T4}}{(\text{T1} \rightarrow \text{T2}) \text{ is a subtype of } (\text{T3} \rightarrow \text{T4})}$$

By the above rule, we can infer that a function f of type T1 → T2 can be used in contexts where functions of type T3 → T4 are expected. In other words, f can accept inputs of type T3 and the outputs of f can appear in contexts where expressions of type T4 are expected.

For example, a function of type Honda → Honda can safely be used wherever a Civic → Car was expected, and a function of type Car → Car can be used anywhere a Honda → Vehicle was expected.

Following code snippet is code interpretation of FunType subtyping rule.

```
override def subType(other:Type): Boolean ={
  if(other==Type.TYPE)
    return true
  if(!other.isInstanceOf[FunType])
    return false;
  val funType = other.asInstanceOf[FunType]
  return funType.getDomain().subType(domain) && range.subType(funType
.getRange())
}
```

Listing 4.6: FunType subtyping

4.2.3 Variable Types

VarType represents Jedi variable type. VarType is composed of variable content type.

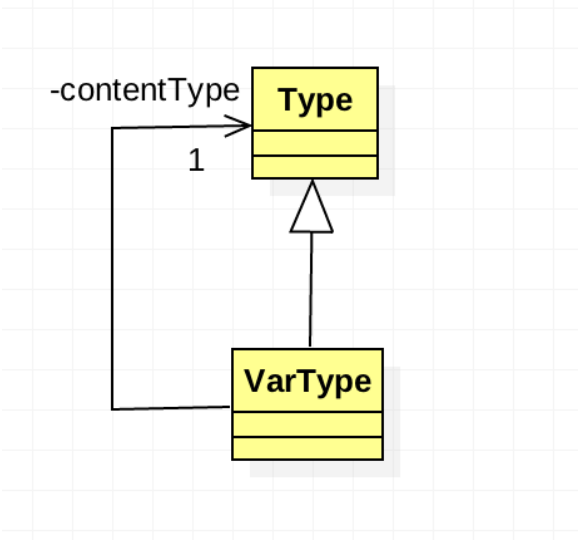

Figure 11: Variable type - class diagram

Following is the code snippet for VarType construct. Refer appendix for complete code.

```
class VarType(contentType: Type) extends Type  { ... }
```

Listing 4.7: VarType class

Formation rule for VarType is,

If T is a type then variable(T) is a type

$$\frac{\text{T is a type}}{\text{variable(T) is a type}}$$

VarType subtyping rule explains, subtyping relation between two VarTypes.

If T is a subtype of S then variable(T) is a subtype of variable(S).

$$\frac{\text{T is a subtype of S}}{\text{variable(T) is a subtype of variable(S)}}$$

Following code snippet is code interpretation of VarType subtyping rule.

```
override def subType(other:Type): Boolean ={
   if(other==Type.TYPE)
     return true
   if(!other.isInstanceOf[VarType])
     return false;
   val varType = other.asInstanceOf[VarType]
   if(!varType.getContentType().toString().equalsIgnoreCase(
 contentType.toString()) && !contentType.subType(varType.
 getContentType()))
       return false
   return true
 }
```

Listing 4.8: VarType subtyping

```
-> def a = var(1)
OK
-> type a
Variable(Number)
-> def fun = lambda(x:Variable(Number)) x
OK
-> fun(a)
Variable(1.0)
-> def b = var(Rat(1,2))
OK
-> type b
Variable(Rational)
-> fun(b)
Variable(0.5)
-> fun(1)
Error
```

Figure 12: Example : VarType subtyping

Example demonstrates, variable(Rational) is masquerading as variable(Number) in lambda function.

CHAPTER 5

Static Type Checking in Jedi

Jedi has a static type system, which infers the type of every expression before any its execution. Jedi architecture is enhanced with new functionalities to add static type checking capabilities.

5.1 Expression

New function to determine the type of expression is introduced in Jedi's Expression. Expression trait is enhanced with 'getType' method, which takes environment as a parameter and infers the type of expression.

```
1 trait Expression {
2   def execute(env: Environment): Value
3   def getType(env: Environment): Type
4 }
```

Listing 5.1: Expression trait

5.2 Literals

Changes made in Literal class to enable static type checking. 'getType' method from expression is inherited and overrode in Literal class.

```
1 case class Literal() extends Expression with Value with Serializable {
2     override def execute(env: Environment): Value = this
3     override def getType(env: Environment):Type = this.typ
4 }
```

Listing 5.2: Literal class

5.3 Identifiers

Identifier and its value are stored in the environment. To implement type checking in Jedi, Identifier construct is changed to hold the type of identifier. So, the environment is maintaining identifier type along with its value.

```
case class Identifier(name: String,var typ: Type =null) extends
    Expression with Serializable {
  def execute(env: Environment): Value = {
    val vals = env.find(this)
    vals
  }
  def getType(env: Environment):Type ={
    val vals = env.find(this)
    vals.getType()
  }
}
```

Listing 5.3: Identifier class

5.4 Conditionals

Conditional operations perform different computations depending on whether a specified boolean condition evaluates to true or false. Conditional expressions are composed of three expressions - condition, consequent, and alternate.

Following is the code snippet for Conditional construct.

```
case class Conditional(conditon: Expression, consequent: Expression,
   alternate: Expression = null) extends SpecialForm { ... }
```

Listing 5.4: Conditional class

Introduction rule for Conditional is,

$$\frac{\text{e1: BOOLEAN, e2: T, e3: T}}{\text{if(e1) e2 else e3 : T}}$$

If expression e1 is of type BOOLEAN and expression e2, e3 are of same type T then if(e1) e2 else e3 is of type T.

Also, **Elimination rule** for Conditional is,

$$\frac{\text{if(e1) e2 else e3 : T, e1: BOOLEAN}}{\text{e2: T, e3: T}}$$

Following is a code interpretation of Conditional rule.

```
1  def getType(env: Environment):Type ={
2    if(conditon.getType(env)==Type.BOOLEAN){
3      if(alternate!=null){
4          if(consequent.getType(env)==alternate.getType(env))
5            consequent.getType(env)
6          else
7            Type.ERROR
8      }else{
9         consequent.getType(env)
10     }
11   }
12   else
13     Type.ERROR
14 }
```

Listing 5.5: Method to infer type of Conditional expression

Example demonstrates, type checking for conditional expressions in Jedi.

```
-> type if(true) 1 + 2 else 3 * 3
Number
-> type if(true) 1 else true
Error
-> type if(1) true else false
Error
```

Figure 13: Example : Type checking of Conditional expressions

5.5 Iterations

Jedi Iterations conducts repetitive computation of a block of statements within a computer program. 'while' expression represents iterations in Jedi. Iteration expressions are composed of a list of expressions with first expression as a condition for while expression.

Following is the code snippet for Iteration construct.

```
1 case class Iteration(operands: List[Expression]) extends SpecialForm {
    ... }
```

Listing 5.6: Iteration class construct

Introduction rule for Iteration is,

$$\frac{\text{e1: BOOLEAN, e2: T}}{\text{while(e1) e2 : T}}$$

If expression e1 is of type BOOLEAN and expression e2 of type T then while(e1) e2 is of type T.

Also, **Elimination rule** for Iteration is,

$$\frac{\text{while(e1) e2 : T, e1: BOOLEAN}}{\text{e2: T}}$$

Code interpretation of Iteration rule.

```
def getType(env: Environment):Type ={
  if(operands(0).getType(env)==Type.BOOLEAN)
    operands(1).getType(env)
  else Type.ERROR
}
```

Listing 5.7: Method to infer type of Iteration expression

```
-> type while(true) 1 + 2
Number
-> type while(1 +2) true
Error
```

Figure 14: Example : Type checking of Iteration expression

Example demonstrates, type checking for iteration expressions in Jedi.

5.6 Blocks

Jedi block allows several expressions to be grouped into a single expression. It enables programmers to define identifiers with restricted scopes and lifetimes.

Following is the code snippet for Block construct.

```
case class Block(locals: List[Expression]) extends SpecialForm { ... }
```

Listing 5.8: Block class construct

Introduction rule for Block is,

$$\frac{\text{e1: T1, e2: T2, ..., en : Tn}}{\{\text{e1; e2; ... ; en}\} : \text{Tn}}$$

If expressions e1 is of type T1, e2 of type T2, ... , en of type Tn then {e1; e2; ... ; en} is of type Tn.

Also, **Elimination rule** for Block is,

$$\frac{\{e1;\ e2;\ \ldots\ ;\ en\} : Tn}{en:\ Tn}$$

Code interpretation of Block rule.

```
def getType(env: Environment):Type ={
  locals.last.getType(env)
}
```

Listing 5.9: Method to infer type of Block expression

```
-> type {def a =1; a = a + 1; if(true) 1 + 2 else 3/4}
Number
-> type {def a =1; a = a + 1; if(true) 1 + 2 else true}
Error
```

Figure 15: Example : Type checking of Block expression

Example demonstrates, type checking for block expressions in Jedi.

5.7 Assignments

Jedi's Assignment statement sets and resets the value stored in the storage location indicated by a variable name. It copies a value into the variable.

Following is the code snippet for Assignment construct.

```
case class Assignment(id: Identifier, exp: Expression) extends
   SpecialForm { ... }
```

Listing 5.10: Assignment class construct

Introduction rule for Assignment is,

$$\frac{e1 : \text{variable(T)},\ e2 : T}{e1 = e2 : \text{NOTIFICATION}}$$

If expressions e1 is of type variable(T), e2 of type T then e1 = e2 is of type NOTIFICATION.

Also, **Elimination rule** for Assignment is,

$$\frac{e1 = e2 : \text{NOTIFICATION}, e1 : \text{variable(T)}}{e2 : T}$$

Code snippet is a interpretation of Assignment rule.

```
def getType(env: Environment):Type ={
  if(exp.getType(env)!=id.getType(env).asInstanceOf[VarType].
 getContentType()){
    Type.ERROR
  }
  else{
    Type.NOTIFICATION
  }
}
```

Listing 5.11: Method to infer type of Assignment expression

```
-> def a = var(2)
OK
-> a = 3
done
-> a
Variable(3.0)
-> [a]
3.0
```

Figure 16: Example : Type checking of Assignment expression

Example demonstrates, type checking for Assignment expressions in Jedi.

5.8 Declarations

Jedi declarations enable the user to declare a variable and use it as a parameter to function or part of the expression.

Following is the code snippet for Declaration construct.

```
case class Declaration(ident: Identifier, exp: Expression) extends
   SpecialForm { ... }
```

Listing 5.12: Declaration class

Introduction rule for Declaration is,

$$\frac{}{\text{e : NOTIFICATION}}$$

Type of declaration expression is always NOTIFICATION.

Code interpretation of Declaration rule.

```
def getType(env: Environment):Type ={
   Notification.OK().getType()
}
```

Listing 5.13: Method to infer type of Declaration expression

```
-> type def x = 2
Notification
-> type def z = x + 1
Notification
```

Figure 17: Example : Type checking of Declaration expression

Example demonstrates, type checking for block expressions in Jedi.

5.9 Function Definitions and Calls

In Jedi, Lambda expressions are a function, and they can be executed like function calls. Function definitions are composed of operator expression, and operand expressions.

Following is the code snippet for FunCall construct.

```
case class FunCall(operator: Expression, operands: List[Expression] =
    Nil) extends Expression { ... }
```

Listing 5.14: FunCall class

Introduction rule for Function definition is,

$$\frac{e : T}{(\text{lambda(x: S) e}): S \rightarrow T}$$

If expressions e is of type T then (lambda(x: S) e) is of type S → T, where x is parameter to lambda expression and x is of type S.

Also, **Elimination rule** for Function call is,

$$\frac{f: S \rightarrow T, e: S}{f(e): T}$$

Code interpretation of rule for function definition and call.

```
    for((m,n) <- tupleType.getComponent().zip(operands.map(_.getType(env
    ))))){

      if(!m.toString().equals(n.toString()) && !n.subType(m)){
        return Type.ERROR
      }
    }
    funType.getRange()
```

Listing 5.15: Method to infer type of function definition and call

Example demonstrates, type checking for function definition and call expressions in Jedi.

```
-> def fun = lambda(x:Number, y:Boolean) if(y) x + 1 else x * 2
OK
-> type fun
((Number * Boolean) -> Number)
-> fun(1,true)
2.0
-> type fun(1,true)
Number
-> type fun(1,1)
Error
```

Figure 18: Example : Type checking of Function definition and call expressions

CHAPTER 6

Conclusion And Future Work

In this thesis work, we surveyed type systems in modern languages and type inference mechanism that allows the programmer to omit certain type annotations. We also studied type polymorphisms and demonstrated our study of such type system by the implementation of the type system in demo language, Jedi.

Our type system implementation includes a runtime type identification system for Jedi values as well as a static type system for Jedi expressions. Also, we successfully implemented inference algorithm to determine types of complex expressions such as lamdas, function calls, and nested expressions. We implemented Jedi type function, with one can inquire for the type of Jedi expression and value. With this, we have achieved a primitive type of reflection in Jedi.

Future work would be an implementation of parametric polymorphism i.e. generic support in Jedi type system. As discussed in our survey, in parametric polymorphism, a function or a data type are generically defined to handle values in a similar way without depending on their type. There is no way to define user define types Jedi, which can be achieved in next version Jedi. Also, Introduction of a class system in Jedi will be a good milestone.

LIST OF REFERENCES

[1] B. Pierce, *Types and programming languages*, Cambridge, Mass.: MIT Press, 2002

[2] K. Bruce, *Foundations of object-oriented languages*, Cambridge, Mass.: MIT Press, 2002

[3] S. Thompson, *Type theory and functional programming*, Wokingham, England: Addison-Wesley, 1991

[4] L. Cardelli and P. Wegner, 'On Understanding Types, Data Abstraction, and Polymorphism', ACM Computing Surveys, vol. 17, no. 4, pp. 471-523, 1985.

[5] A. Gupta, 'So You Still Don't Understand Hindley-Milner? Part 3 - Amit's Blog', Amit's Blog, 2013, `http://akgupta.ca/blog/2013/06/07/so-you-still-dont-understand-hindley-milner-part-3/`

[6] L. Damas and R. Milner, 'Principal type-schemes for functional programs', ACM, pp. 207–212, 1982.

[7] Book.realworldhaskell.org, 'Chapter 2. Types and Functions', 2015, `http://book.realworldhaskell.org/read/types-and-functions.html`

[8] Docs.scala-lang.org, 'Introduction - Scala Documentation', 2015, `http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html`

[9] Bjarne Stroustrup, *The C++ Programming Language*, 4th ed. Wesley. Reading Mass, 2013

[10] Cs.sjsu.edu, 2015 `http://www.cs.sjsu.edu/faculty/pearce/modules/projects/Jedi/index.htm`

APPENDIX

Following is the code snippet for class Value.

```
trait Value extends Serializable {
  var typ: Type = Type.VALUE;
  def getType():Type = { return typ }
}
```

Listing A.1: Jedi Value

Following is the code snippet for class TupleType.

```
class TupleType(components: List[Type]) extends Type {
    this.typ = Type.TUPLE;
    override def toString(): String ={
      "("+components.mkString(" * ")+")"
    }
    def getComponent(): List[Type] ={
      components
    }
    override def subType(other:Type): Boolean ={
     if(other==Type.TYPE)
       return true
     if(!other.isInstanceOf[TupleType])
       return false;
     val tupleType = other.asInstanceOf[TupleType]
     if(tupleType.getComponent().size==components.size){
        for((m,n) <- components.zip(tupleType.getComponent())){
          if(!m.toString().equals(n.toString()) && !m.subType(n)){
            return false
          }
```

```
        }
      }
      else return false
    return true
  }
}
```

Listing A.2: Jedi TupleType

Following is the code snippet for class FunType.

```
class FunType(domain: Type, range:Type) extends Type{
  this.typ = Type.FUNCTION;
   override def toString(): String ={
      "("+domain+" -> "+range+")";
    }
   def getDomain(): Type ={
      domain;
    }
   def getRange(): Type ={
      range;
   }
   override def subType(other:Type): Boolean ={
     if(other==Type.TYPE)
       return true
     if(!other.isInstanceOf[FunType])
       return false;
     val funType = other.asInstanceOf[FunType]
     return funType.getDomain().subType(domain) && range.subType(funType
    .getRange())
   }
```

```
}
```

Listing A.3: Jedi FunType

Following is the code snippet for class VarType.

```
class VarType(contentType: Type) extends Type  {
  this.typ = Type.VARIABLE;
  override def toString(): String ={
      this.typ+"("+contentType.toString()+")";
  }
  def getContentType(): Type ={
    contentType
  }
  override def subType(other:Type): Boolean ={
     if(other==Type.TYPE)
       return true
     if(!other.isInstanceOf[VarType])
       return false;
     val varType = other.asInstanceOf[VarType]
     if(!varType.getContentType().toString().equalsIgnoreCase(
   contentType.toString()) && !contentType.subType(varType.
   getContentType()))
         return false
     return true
   }
}
```

Listing A.4: Jedi VarType

Following is the code snippet for class Iteration.

```
case class Iteration(operands: List[Expression]) extends SpecialForm {
  def execute(env: Environment): Value = {
```

```
3     if (operands.length != 2) throw new TypeException("iteration expects
      exactly 2 operands")
4     val operand0 = operands(0).execute(env)
5     val ok = operand0.isInstanceOf[Boole]
6     if (!ok) throw new TypeException("first iteration operand must
    evaluate as boole")
7     while (operands(0).execute(env).asInstanceOf[Boole].value == true) {
8       operands(1).execute(env)
9     }
10    Notification.DONE
11  }
12  def getType(env: Environment):Type ={
13    if(operands(0).getType(env)==Type.BOOLEAN)
14      operands(1).getType(env)
15    else
16      Type.ERROR
17  }
18 }
```

Listing A.5: Jedi Iteration

Following is the code snippet for class Conditional.

```
1 case class Conditional(conditon: Expression, consequent: Expression,
    alternate: Expression = null) extends SpecialForm {
2     def execute(env: Environment): Value = {
3       var check = conditon.execute(env)
4       if (!check.isInstanceOf[Boole]){
5         throw new TypeException("Condition needs to be boolean type")
6       }
7       var result = check.asInstanceOf[Boole]
8       if (result.value){
```

```
9           consequent.execute(env)
10        }
11        else{
12          if (alternate != null)
13            alternate.execute(env)
14          else
15            throw new TypeException("Type exception")
16        }
17      }
18    def getType(env: Environment):Type ={
19      if(conditon.getType(env)==Type.BOOLEAN){
20        if(alternate!=null){
21            if(consequent.getType(env)==alternate.getType(env))
22              consequent.getType(env)
23            else
24              Type.ERROR
25        }else{
26           consequent.getType(env)
27        }
28      }
29      else
30        Type.ERROR
31    }
32  }
```

Listing A.6: Jedi Conditional